ARPA ORDER NO. 2223

ISI/RR-77-62
October 1977

David R. Musser

A Proof Rule for Functions

INFORMATION SCIENCES INSTITUTE
UNIVERSITY OF SOUTHERN CALIFORNIA
4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) ISI/RR-77-62	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) A Proof Rule for Functions,		5. TYPE OF REPORT & PERIOD COVERED (9) Research rept.,
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) David R. Musser		8. CONTRACT OR GRANT NUMBER(s) (15) DAHC15-72-C-0308
9. PERFORMING ORGANIZATION NAME AND ADDRESS USC/Information Sciences Institute 4676 Admiralty Way Marina del Rey, CA 90291		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA Order 2223 Program Code 3D30 & 3P10
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		(11) 12. REPORT DATE October 1977 13. NUMBER OF PAGES 10
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ----- (12) 10p.		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

This document approved for public release and sale; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

-----

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

functions, program verification, proof rules

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report gives a rule of inference which permits a natural form of reasoning about programs containing functions. The rule was obtained by modifying the function rule of Clint and Hoare to include a premise which requires consistency of preconditions and postconditions, and a premise which requires deterministic computation. Examples of use of the rule and versions tailored to the Pascal and Euclid languages are also given.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407952

AB

ARPA ORDER NO. 2223

ISI/RR-77-62
October 1977

David R. Musser

A Proof Rule for Functions

INFORMATION SCIENCES INSTITUTE

UNIVERSITY OF SOUTHERN CALIFORNIA

4676 Admiralty Way/ Marina del Rey/ California 90291
(213) 822-1511

THIS RESEARCH IS SUPPORTED BY THE ADVANCED RESEARCH PROJECTS AGENCY UNDER CONTRACT NO. DAHC15 72 C 0308, ARPA ORDER NO. 2223 PROGRAM CODE NO. 3D30 AND 3P10.

Introduction

Clint and Hoare [2] gave a proof rule for functions (without side effects) which later was shown by Ashcroft [1] to be unsound. The way of avoiding the unsoundness suggested in [1] seems unattractive in that it requires "an unconventional interpretation for functional notation," while other proof rules that have been proposed for functions are more complicated (e.g., as in [7]) or require proofs of termination for functions. This paper suggests a simple modification to Clint and Hoare's proof rule--an additional "consistency premise"--which makes it sound without requiring proofs of termination. Although proving termination is in most cases a worthy goal, one may wish to consider functions that fail to terminate because they exit abnormally by a jump to an external label [2]. For such functions the present proof rule will be especially useful. It also permits a simpler algorithm for generating verification conditions than that described in [7], and has been implemented by the author in a verification condition generator at Information Sciences Institute.

The new proof rule, like the Clint and Hoare rule, allows (and requires) the result of the computation performed by the body of a given function f to be specified as a mathematical function[1] of the input values, using the name f as the name of the mathematical function also. In order to make the proof rule applicable to programming languages in which nondeterministic computations are expressible, the rule includes a premise requiring that the body of the function deterministically computes its result value.

Following some simple examples of use of the new proof rule, we give versions of the rule adapted to the Pascal [6] and Euclid [9,11] languages, and conclude with a discussion of the role in programming languages of function-like constructs which do have side effects.

Notation

Hoare's notation P{S}R, where P and R are predicates and S is a statement in a programming language, will be used to express the assertion that if P is true before the execution of S then R is true after the execution of S. P{S}R is vacuously true if S fails to terminate. Rather than considering this to be a different kind of assertion from an ordinary predicate, we will assume P{S}R is defined as a predicate transformer; e.g., in terms of the predicate transformer wlp ("weakest liberal precondition") defined in [4], we have P{S}R ≡ (P ⊃ wlp(S,R)). Thus the usual logical operators, including quantification, may be used in conjunction with these assertions.

The notation

$$P_{y \to x}$$

will be used to denote the predicate which is obtained by substituting y for all free occurrences of x in P.

---

[1] Throughout this paper the term "function" will refer to a programming language concept and the term "mathematical function" will refer to the function concept in ordinary mathematical usage, i.e., an association of exactly one element from one set with each element of another set.

Thus, for example, $P\{x:=e\}R \equiv (P \supset R_{e\to x})$.

The proof rule

A function is declared by the schema

function *f(x)* **returns** *z; S*

where f is the function name, S is the body of the function, x is a list of formal parameters including all of the free non-local variables appearing in S, and z is the return variable (which is assigned to by one or more statements in S). It is assumed that S makes no assignment to any of its parameters so that there is no possibility of side effects. Let P and R be predicates (P will be referred to as a precondition and R as a postcondition for f; generally both P and R will depend on x and R will also depend on z). From

(1) [Consistency] $\forall x \exists z(P \supset R)$

(2) [Determinism] $\forall x \exists z1(P\{S\}z=z1)$

and

(3) [Property of S] $\forall x(P\{S\}R)$

we may deduce the following implication:

(4) [Property of f] $\forall x(P \supset R_{f(x)\to z})$.

This property (4) may be assumed in proving assertions about expressions containing calls of the function f, including those occurring within S itself. If other properties of the function f have been added to the proof system previously, then premise (1) should be replaced by

(1') [Global consistency] $\forall x \exists z(Q \wedge (P \supset R))$

where Q is the conjunction of all properties of f with all occurrences of f(x) replaced by z.

Discussion

This proof rule differs from the Clint and Hoare rule mainly in the addition of (1) (or (1')) and (2) as premises. The role of (1) and (1') is to prevent the use of a postcondition that is so strong that no mathematical function could exist satisfying (4). If each function has associated with it a single precondition and single postcondition, as for example in the Euclid language, then use of (1') would be unnecessary. However, one may wish to state and prove properties of a function incrementally, or by means of a collection of "axioms" about a whole set of functions, as in the algebraic axioms method of specification of an abstract data type [5]. In this case (1') should be used.

Premise (2) requires that the value of z computed by S is some mathematical function of x; i.e., S computes z deterministically from x. Without this requirement it would be possible to have computations denoted by functional notation even though the result of the computation is indescribable as a mathematical function of its inputs. For languages in which every computation is deterministic, such as Pascal, this premise is trivially satisfied and can be ignored. The Euclid language, however, permits a form of nondeterminism, as will be discussed below.

If S contains (recursive) calls of f, then property (4) should be added to the proof system before attempting to prove (3), so that (4) may be used in the proof of (3). Thus as a general rule, first premise (1) (or (1')) should be established without using (4), then (4) may be added to the proof system, then the proof of (3) may be attempted. If (3) is not provable, then either S should be modified or (4) should be deleted from the proof system.

Examples

In [1] the following example was given showing the unsoundness of the original rule, which had only the property of S, (3) as a premise:

function f(x:integer) **returns** z:integer;
begin z:=0; **while true do null end**
P: TRUE
R: FALSE $\wedge$ z=0

It is possible to prove (3) (taking z=0 as the invariant of the **while** loop), which would give $\forall x(\text{TRUE} \supset (\text{FALSE} \wedge f(x)=0))$ or simply FALSE for the property of f, so that the addition of this property to the proof system would make it inconsistent. For this P and R, however, the consistency premise (1) is $\forall x \exists z(\text{TRUE} \supset (\text{FALSE} \wedge z=0))$, which reduces to FALSE. Thus the requirement of premise (1) serves to prohibit the addition of (4) to the proof system in this case.

If R in this example were weakened to be just z=0, then the consistency premise (1) would be $\forall x \exists z(\text{TRUE} \supset z=0)$, which is trivially provable, and the property $\forall x(\text{TRUE} \supset f(x)=0)$, or simply $\forall x(f(x)=0)$, could be added to the proof system. The fact that the body of the function always fails to terminate makes it rather useless, but at least no unsoundness can arise from use of the property $\forall x(f(x)=0)$ in any proofs.

To see how the consistency premise (1) comes into play in the verification of a useful function, consider the example given in [2], a function *lookup(A,N,x)* which searches for an element x in a sorted array A of length N, returning an index m such that A[m]=x if x is contained in A, and jumping out to an external label if not. If we take as pre- and postconditions for *lookup*[2]

P: $1 \leq N \wedge$ sorted(A,N)
R: A[m]=x

---

[2] These were the pre- and postconditions used in [2], except for some inessential details.

then the consistency premise would be $\forall A,N,x \exists m(1 \le N \wedge sorted(A,N) \supset A[m]=x)$, which is false. However, if R is weakened to be $\exists i(1 \le i \le N \wedge A[i]=x) \supset A[m]=x$, then the resulting consistency premise is true (and is trivial to prove). Since the weaker postcondition is a more accurate description of the behavior of the function, the requirement of the consistency premise seems to support the use of pre- and postconditions to document programs in a precise manner, aside from its role in preventing inconsistency.

A simple example of the necessity of using the global consistency premise (1') instead of (1) in the case of multiple pre- and postconditions is the following:

function f(x:integer) **returns** z:integer;
begin if x=0 **then while true do null else** z:=x **end**
P_1: $x \ge 0$
R_1: $z > 0$
P_2: $x \le 0$
R_2: $z < 0$

From P_1 and R_1 we obtain the property $\forall x(x \ge 0 \supset f(x) > 0)$ and from P_2 and R_2 we would obtain the contradictory property $\forall x(x \le 0 \supset f(x) < 0)$ if only premise (1) were required. However, (1') is $\forall x \exists z((x \ge 0 \supset z > 0) \wedge (x \le 0 \supset z < 0))$, which is false, prohibiting the adoption of the second property.

To illustrate the role of the determinism premise (2), suppose that the construct S1 **or** S2 means that either statement S1 or statement S2 is to be executed, the choice being made nondeterministically [10]. Formally,

$$P\{S1 \textbf{ or } S2\}R \equiv P\{S1\}R \wedge P\{S2\}R;$$

in particular,

$$P\{x:=e1 \textbf{ or } x:=e2\}R \equiv (P \supset R_{e1 \to x} \wedge R_{e2 \to x}).$$

Then if we attempt to write

function f(x:integer) **returns** z:integer;
begin z := 0 **or** z := 1 **end**;

we obtain as premise (2)

$$\forall x \exists z1(P \supset (z1=0 \wedge z1=1))$$

which is false for any satisfiable P. Therefore this nondeterministic assignment to z could not be used as a function body.

Pascal functions

The proof rule given in [6] for Pascal functions was based on the rule of [2] and has the same problem of unsoundness. The following differs from the rule given in [6] in the addition

of the consistency premises (P1) and (P1') and in permitting postconditions to refer to initial values of parameters (parameters to functions in Pascal are value-parameters).

A function is declared by the schema

function *f(L):T; S*

where L is a list of identifiers and types, T is a type name (the type of the return value of the function, for which the name of the function, f, is used), and S is a statement. Let x be the list of parameters declared in L, and let y be the list of nonlocal variables occurring within S (implicit parameters). Given predicates P and R, where f does not occur free in P and none of the variables of x occurs free in R (and occurrence in R of primed variables x' denotes initial values of the parameters x), then from

(P1) [Consistency] $\forall x,y \exists f (P \supset R_{x \to x'})$

and

(P2) [Property of S] $\forall x,y,x' ((x=x' \wedge P)\{S\}R)$

we may deduce the following implication:

(P3) [Property of f] $\forall x,y (P \supset R_{f(x,y) \to f,\ x \to x'})$.

Note that the explicit parameter list x has been extended by the implicit parameters y, that x may not contain any variable parameters (specified by var*) and that no assignments to nonlocal variables may occur within S. It is property (P3) that may be assumed in proving assertions about expressions containing calls of the function f, including those occurring within S itself and in other declarations in the same block. In addition, assertions generated by the parameter specifications in L may be used in proving assertions about S. If other properties of the function f have been added to the proof system previously, premise (P1) should be replaced by*

(P1')[Global consistency] $\forall x,y \exists f (Q \wedge (P \supset R_{x \to x'}))$

where Q is the conjunction of all properties of f with all occurrences of f(x,y) replaced by f.

Euclid functions

In the Euclid language, the definition of functions is complicated by the possibility of nondeterministic behavior of operations defined in a Euclid module. The *abstraction function* of a module implicitly defines an equality relation on values of the type defined by the module; with respect to this equality, operations of the module may be nondeterministic since they may behave differently for different concrete representations of the same (abstractly equal) inputs. Thus the premise requiring deterministic computation is included in the following proof rule for Euclid functions.

A function is declared by the schema

function *f(L)* **returns** *z:T* = **imports**(*M*); **pre** *P*; **post** *R*; *S*

where L and M are lists of identifiers with types, z is the return variable, T is a type, P and R are predicates, and S is a statement. Let x be the list of parameters declared in L and y be the list of parameters declared in M. From

(E1) [Consistency] $\forall x,y \exists z (P \supset R)$

(E2) [Determinism] $\forall x,y \exists z1 (P\{S\}z=z1)$

and

(E3)[Property of S] $\forall x,y (P\{S\}R)$

we may deduce the following implication:

(E4) [Property of f] $\forall x,y (P \supset R_{f(x,y)\rightarrow z})$.

Note that the definition of Euclid does not permit any of the parameters x or y to be altered (they are **const** *or* **readonly**). *It is property (E4) that may be assumed in proving assertions about expressions containing calls of the function f, including those occurring within S itself and in other declarations in the same block. In addition, the axiom R{***return***}FALSE and assertions generated by the specifications in L and M may be used in proving assertions about S.*

Conclusions

We have given a proof rule for functions without side effects, essentially the Clint and Hoare rule modified with some additional premises. These additions eliminate the possibility of unsoundness and permit reasoning about functions to be carried out in familiar mathematical notation even in cases in which the body of the function does not terminate for some inputs. In many common programming languages, what are called functions have no strictures against side effects and thus are not characterizable by the simple proof rule we have given. Proof rules which have been proposed for such "functions" [3,8] effectively treat them as a special kind of procedure call and require the introduction of new symbols to denote return values. In languages such as Pascal and Euclid, any computations having side effects would have to be programmed as procedures, and thus the proof rule for procedure calls would apply directly. In most situations, this would seem to be the most satisfactory way of programming and reasoning about computations, as it provides a clear distinction between true functions and computations having side effects. It is difficult, however, to argue that *all* computations which have traditionally been programmed as "functions" even though they have side effects should instead be programmed as procedures. The Lisp function CONS, for example, has a side effect on the free storage list, and thus all Lisp functions that use CONS have side effects. In the design of future programming languages it will probably be necessary to retain some sort of value-returning procedure (callable from expressions), but it would seem worthwhile also to have a pure function construct to which the proof rule discussed in this paper would be applicable. Then those computations which could be programmed without side effects could still be characterized via the simple concepts of mathematical functions.

Acknowledgments

I wish to thank my colleagues at ISI for helpful comments on earlier versions of this paper. I am especially grateful to John Guttag and Ralph London for a number of stimulating discussions and useful suggestions.

REFERENCES

1. Ashcroft, E.A., Clint, M., and Hoare, C.A.R.: Remarks on "Program proving: Jumps and functions by M.Clint and C.A.R.Hoare". Acta Informatica 6, 317-318 (1976)

2. Clint,M., and Hoare, C.A.R.: Program proving: Jumps and functions. Acta Informatica 1, 214-224 (1972)

3. Cunningham, R.J. and Gilford, M.E.J.: A note on the semantic definition of side effects. Information Processing Letters 4, 118-120 (1976)

4. Dijkstra, E.W.: A discipline of programming. Englewood Cliffs, New Jersey: Prentice-Hall 1976

5. Guttag, J.V., Horowitz, E., and Musser, D.R.: Abstract data types and software validation, USC Information Sciences Institute Report ISI/RR-76-48 (August 1976)

6. Hoare, C.A.R., and Wirth, N.: An axiomatic definition of the programming language Pascal. Acta Informatica 2, 335-355 (1973)

7. Igarashi, S., London, R.L., and Luckham, D.C.: Automatic program verification I: A logical basis and its implementation. Acta Informatica 4, 145-182 (1975)

8. Kowaltowski, T.: Axiomatic approach to side effects and general jumps. Acta Informatica 7, 357-360 (1977)

9. Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J.: Report on the programming language Euclid. SIGPLAN Notices, 12, 2 (1977)

10. Lauer, P.: Consistent formal theories of the semantics of programming languages. IBM Laboratory, Vienna, TR 25.121 (November 1971)

11. London, R.L., Guttag, J.V., Horning, J.J., Lampson, B.W., Mitchell, J.G., and Popek, G.J.: Proof rules for the programming language Euclid. Technical Report (May 1977)